(4)

AD-A229 055

RADC-TR-90-203, Vol III (of three)
Final Technical Report
September 1990

DOS DESIGN/APPLICATION TOOLS System/Segment Specification

Honeywell Corp.

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

90 11 27 034

This report has been reviewed by the RADC Public Affairs Division (PA) and is releasable to the National Technical Information Services (NTIS) At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-90-203, Vol III (of three) has been reviewed and is approved for publication.

APPROVED:

THOMAS F. LAWRENCE
Project Engineer

APPROVED:

RAYMOND P. URTZ, JR.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER:

IGOR G. PLONISCH
Directorate of Plans & Programs

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY *(Leave Blank)*

2. REPORT DATE

3. REPORT TYPE AND DATES COVERED
Final Dec 87 - Dec 89

4. TITLE AND SUBTITLE
DOS DESIGN/APPLICATION TOOLS System/Segment Specification

5. FUNDING NUMBERS
C - F30602-87-C-0104
PE - 62702F
PR - 5581
TA - 21
WU - 78

6. AUTHOR(S)

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Honeywell Corporation
Sensor and System Development Center
1000 Boone Ave, North
Golden Valley MN 55427

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Rome Air Development Center (COTD)
Griffiss AFB NY 13441-5700

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
RADC-TR-90-203, Vol III (of three)

11. SUPPLEMENTARY NOTES
RADC Project Engineer: Thomas F. Lawrence/COTD/(315) 330-2158

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

date

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Developing applications for execution in a distributed processing environment is a difficult task. Such environments dominate Air Force C^3I systems, which are necessarily distributed. In addition to being a physical necessity, distributed systems offer, relative to centralized processing systems, the potential for increased performance and fault tolerance. Realizing that potential is a key objective behind research in distributed systems technology.

The goal of this contract is to:
1) Define and demonstrate a framework for integrating development tools.
2) Define and construct tools that support the development of distributed applications.

A tool integration platform was designed and developed as a fundamental element of an integrated development framework. The RADC Distributed System Evaluation (DISE) Environment Tool Integration Platform integrates software development tools by automating and coordinating information exchange between tools, through use of the CRONUS distributed system and the ONTOS object oriented database management system. (Continued)

14. SUBJECT TERMS
Software Development Tools, Resource Allocation, Tool Integration, Reliability Analysis, Distributed System, Object Oriented DBMS

15. NUMBER OF PAGES
44

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (890622)
Prescribed by ANSI Std. Z39-18
298-01

Block 13 (Continued)

Two development tools were selected and implemented that illustrate the types of technology required to support distributed application development. The Allocator assists developers with determining efficient implementations for distributed applications. The Reliability Analyzer generates reliability measures for application components given a set of hardware reliabilities. The two tools have been integrated into the IP. (KR)

This report summarizes the contract's objectives and results.

Accession For	
NTIS GRA&I	☑
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	

Dist	Avail and/or Special
A-1	

Table of Contents

List of Figures

1. Scope

1.1. Identification

This system specification establishes the requirements for the Tool Integration Framework and the allocation and reliability analysis tools. They are collectively be referred to as the *System* within this document.

1.2. Purpose

The System consists of the DISE Tool Integration Framework, an allocation tool and a reliability analysis tool. The DISE Tool Integration Framework integrates software development tools for developing distributed applications by automating:

- Data exchange and sharing among technical and project management tools;
- Capture and retention of all information used by any tool or person during software development.

The definition of a software process that controls and disciplines development activities accompanies the System but is not part of the software.

The allocation tool assists developers in determining an efficient assignment of application components (objects) to processing nodes. The reliability analysis tool assists developers with design time development and evaluation of reliable distributed applications.

1.3. Introduction

This document specifies the functional, interface and performance requirements for the DISE Integrating Framework, the allocation tool and the reliability analysis tool. It also specifies, where applicable, the requirements for the characteristics, logistics, quality factors, design, qualification and delivery of the Integration Framework and the tools.

2. Applicable Documents

Interim Technical Report #1, DOS Design Application Tools, Honeywell Inc., Sensor and System Development Center, July 14, 1988.

Software Top-Level Design Document, DOS Design Application Tools, Honeywell Inc., Sensor and System Development Center, March 15, 1988.

Interim Technical Report #2, DOS Design Application Tools, Honeywell Inc., Sensor and System Development Center, June 21, 1989.

3. Requirements

3.1. System Definitions

3.1.1. Missions

A tool integration framework and software development tools do not perform a mission in the strictest sense. They do, however, perform functions in order to fulfill their objectives. In this subsection, "mission" is interpreted to mean objectives in the context of development environments for distributed systems and in particular the DISE environment.

Air Force C^2 applications are large, complex distributed systems that may manipulate very large databases, and exhibit time-dependent, non-replicatable behavior. To perform their missions, they may have stringent performance and reliability constraints. Developing such applications requires all the tools and methods used to develop centralized systems, plus methods and tools to solve problems

unique to the distributed application domain. Developers faced with the challenge of producing such applications need a rich tool set and an integrating framework that makes the tools as convenient as possible to use. In addition, they need an articulated software process that controls and disciplines development. Such a process increases the likelihood that developers produce the desired application within the budgeted resources.

Our system has two main objectives:

- to develop and demonstrate elements of an integrating framework for software tools,
- to provide tools that solve problems unique to the distributed application domain.

It is well known [Penedo88a] that in an integrated environment, automated development tools ease developers' burdens and lead to the production of more reliable software. Integration assures easy transfer of information among tools, maintains consistency of information as the application under development is transformed from requirements to implementation, provides a uniform interface for user interactions with tools, and off-loads menial tasks from people to computers. CASE technology for developing centralized information processing systems is already available on PCs at almost nominal costChikof88a. The benefits of integration should also be available to developers of distributed systems.

An *integrated* Software Engineering Environment (ISEE) supports cooperation among tools and exhibits certain uniformities from a user's point of view. The complete ISEE provides:

- A technical and management software process, tailorable on a per project basis;
- Automated guidance for or enforcement of both the project management and the technical processes;
- Methods for accomplishing every activity in the management and technical processes, and tool support for the methods;
- A uniform user interface for all interactions between users and the ISEE;
- Automated data exchange and sharing among all project management and technical tools;
- Automated capture and retention of all information used by any tool or person during development;
- Extensibility that permits new tools and/or data types to be introduced into the environment during a single development project.

Integration is achieved with the software processes (which bring coherence to the collection of development activities), the uniform user interface and automated data exchange and sharing. An ISEE's *integrating framework* provides principles and automation to implement cooperation among tools and uniformity of tool invocation and data access from a user's point of view.

This System provides these elements of an integrating framework:

- A technical software process that is in reality a meta-process that project members tailor as they progress through the steps of the project.
- An extensible tool integration platform (a project database) in to automate data exchange and sharing among technical tools.

Experience in developing centralized software has established that using a software engineering process greatly improves the chances of attaining the desired system within resource constraints. Software processes are part of the integrating framework; they define the development tasks, control the order among them, and suggest appropriate methods and tools for each task and for the transitions among tasks.

Software development processes are defined with drivers in mind. For example, the major driver behind the waterfall life cycle [Boehm83a] is the generation of work products: requirements, designs, source code, object code, documentation, and so forth. Therefore the tasks it prescribes, the methods and tools for accomplishing those tasks, and the order among tasks are aimed at producing work products.

A key aspect of distributed system development is risk. Therefore, we recommend the Spiral Development Process [Boehm86a] whose major goal is controlling risk through tasks and tools for risk analysis and resolution. The Allocation and Reliability Analysis tools support experiments with software designs to analyze performance and reliability characteristics, allowing designers to tune the software during design, thereby managing the risk that the implemented software will not meet its performance or reliability requirements.

All software development produces and uses masses of information. Management and full exploitation of this information is a key to cost effectively developing software that achieves desired system goals. The major information management issues are:

- Consistency,
- Ability of tools that produce and/or consume the same information to obtain that information without "manual" translation by people.

Solving the information management problem effectively integrates tools already in an environment and prepares the environment to absorb new tools.

DISE already contains tools; for example, Cronus gendoc, genmgr and tropic. While existing tools may not be able to use all the features of all the information in the project database, they must not be excluded from using or contributing to that information somehow. The integrating framework is of use to existing tools. Its main goal, however, is to integrate new tools. Therefore, part of the integrating framework system element is a definition of the requirements on new tools that allow them to take full advantage of the project database.

Because distributed system development involves more experimentation than centralized development, it produces and consumes even more information than centralized development. Distributed systems cannot be developed successfully by experiment without automation for information management. Therefore, we propose a project database as the DISE tool integration infrastructure. This repository contains all relevant information about the system. Consistency is maintained because every stored and derivable item of information is written only once and because the semantics of that item is established by the repository for all tools that access it. Information is available to all tools that need it without human intervention because all tools consume input from and produce output to the same repository. New tools are easily absorbed into the environment; they use the repository either directly or with appropriate tool adapters.

The integrating framework hosts a tool set for distributed development. The System defined in this document contains two tools: an Allocation tool and a Reliability Analysis tool. These tools:

- Support the risk-driven experiment-and-evaluate development process;
- Demonstrate the integrating framework;
- Establish the requirements on tools that are to be integrated into our framework.

Because distributed applications are less well understood than centralized applications they often benefit from being developed iteratively and experimentally. Therefore, an important class of tools for developing distributed applications are those that support risk management experiments with the software under development. A *risk* is defined as an aspect of the application under development that is perceived as a particular threat to meeting the application objectives. The threat is defined in terms

of the potential consequences; i.e., possible deviation from objectives resulting from inadequate treatment of the application aspect. The Allocation and Reliability Analysis tools allow application developers to experiment with software designs before investing in an expensive implementation which fails to meet stated objectives. The Allocation tool assists with the development and implementation of efficient distributed applications in an environment where performance objectives represent a significant challenge to application developers. The Reliability Analysis tool attempts to mitigate the problem of developing distributed applications that meet reliability objectives in faulty processing environments.

3.1.2. Threat

Not applicable.

3.1.3. System Modes and States

For the purposes of this System, modes are defined in terms of the intended user communities for the System functions. Four user modes are relevant in this System:

- Application Developer: An Application Developer develops distributed applications in DISE. This is the user class targeted for the Tools functions in that they are the tool users.
- Tool Builder: The Tool Builder develops tools which support the task of developing distributed applications. These technical tools are built and used in DISE. The DISE Tool Integration Framework function supports this user class.
- Tool Integrator: The Tool Integrator is a user who integrates tools, whether new or existing, into the framework in DISE. In the case of new tools, the Tool Builder is also the Tool Integrator. This user class relies on the DISE Tool Integration Framework function.
- Integration Platform Administrator: This user maintains the DISE Tool Integration Framework on the basis of Application Developer's and Tool Developer's needs.

3.1.4. System Functions

There are three System functions defined in this paragraph: Integration of Tools, the Allocation tool, and the Reliability Analysis tool.

3.1.4.1. Tool Integration Framework System Function

3.1.4.1.1. Rationale

Tool integration assures easy transfer of information among tools, maintains consistency of information as the application under development is transformed from requirements to implementation, provides a uniform interface to user interactions with tools, and off-loads menial tasks from people to computers. This technology for developing centralized information processing systems is already available, and the benefits of integration should also be available to developers of distributed systems.

The objective for this task is to develop a framework for integrating tools in RADC's evolving DISE environment. The framework supports the most effective application of DISE tools, present and future, planned and as yet unknown. DISE is an evolving environment that contains conventional development tools (compilers, linkers, editors and tools for experimenting with applications under development like debuggers, monitors, performance analyzers, etc.). DISE users are sophisticated computer scientists and that tools not conceived of today will be inserted into it. Our integrating framework must accept sophisticated unknown tools as well as conventional development tools. Moreover, most of these tools will be independently developed. Our integrating technology applies to more than DISE, but is tailored to the requirements of DISE.

Hence the two major influences on our framework are: the distributed nature of the software whose development it supports, and the fact that it is intended for the DISE environment. The framework elements and the development tools they support must all contribute to mitigating the risks of developing distributed software. The integrating technology provided by this System will be used first in RADC's DISE.

The purpose of the integrating framework is to (1) support and encourage cooperation among (existing and new) independently developed programming tools by automating (or partially automating) data exchange and sharing among the tools; and (2) capture and retain all information required to develop the application/system (information generated during system development).

3.1.4.1.2. Functionality

The tool integration framework consists of:

- A technical software process that is in reality a meta-process that project members tailor as they progress through the steps of the project.
- Automated data exchange and sharing among tools via a *project database.*

It has been established that using a software process to guide the development process greatly improves the chances of attaining the desired system objectives within resource constraints. Software processes are part of the integrating framework; they define the development tasks, control the order among them, and suggest appropriate methods and tools for each task and for transitions among tasks. This System does not automate enforcement of the software process or provide an assistant that guides the development process.

The integration of tools in the framework is supported by a project database, or "Tool Integration Platform" (IP). Tools in the system interface with the database for all their input/output requirements. This IP serves both as a repository of information/data and as the medium for information exchange among tools. The major components of this integrating framework are the information model, the database schema, the data definition and manipulation languages, the IP and operations to permit data access for the indirectly coupled existing tools. Figure 1 shows the overall tool integration architecture. These components are introduced below and are discussed in detail in subsection 3.1.6.1.1.

Information model: The information model is a list of all the types of data that can appear in the database. Data types are included in the model because they are used as input and output by existing tools or because they are relevant to distributed system development and may be used by some future tool. The model is extensible because the full spectrum of data types needed by all future tools in this evolving environment and supported tool set cannot be predicted in advance.

Database Schema: Data in the database is structured by a schema that implements the information/data model. Database schemas specify the logical structure of databases. The IP is instantiated on a per-project basis; each IP has its own schema consisting of types from the information model. The schema used in the IP supports the tools currently in DISE, the development tools, and the standard software development tools such as a compiler and editor. The schema should:

- Permit different kinds of software entities to be modeled as objects;
- Allow arbitrary relations of any cardinality among these objects;
- Provide methods - that behave consistently - that the tools/data adapters can invoke to access and manipulate the data in the database;
- Structure the different objects/types in a hierarchy to facilitate inheritance.

Data Modeling approach and Data Definition Language (DDL): The DDL is a language (primitives and operations) used to define and extend both the information model and the IP schemas. To tailor

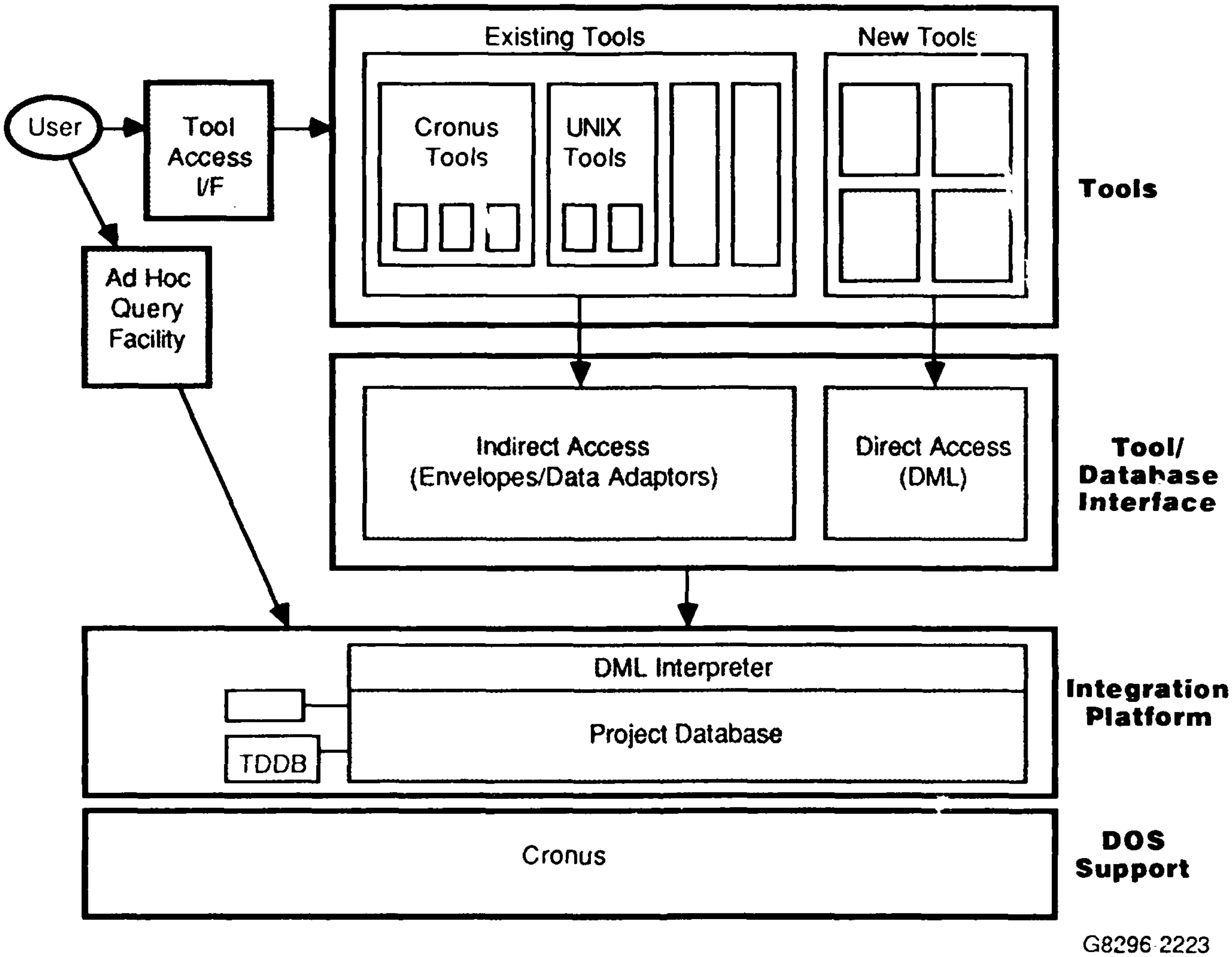

Figure 1. Tool Integration Architecture

the database to the project's specific needs, the user/developer should be able to define and extend the information and data models. Hence, this language should allow:

- The addition of new types/objects to the database schema and extensions to the information model;
- The addition of methods/properties to types;

- The manipulation of schemas to insert new types in the hierarchy;
- The modification of a schema for a project.

Data Manipulation Language (DML): A DML allows a program using data from the database to create, update and delete data according to the given schema. The DML can be embedded in the code of the software development tools. This DML is used to:

- Read and update data objects in the database;
- Delete and create objects in the schema;
- Create and delete relations between objects in the database;
- Specify/determine properties/attributes of the objects.

With an appropriate interface, this language can also be used for ad hoc database queries.

A DML Interpreter in the IP interprets these calls and route the request to the correct object.

Database: The project database acts as the integrating *mechanism*. Unlike a file system, which also acts as a repository of information, the database:

- Explicitly maintains relations between related software entities;
- Provides a consistent view of the data (a canonical form) to the tools;
- Structures all the data associated with a project into a project-specific schema;
- Maintains data consistency and integrity;
- Provides mechanisms for accessing and manipulating the data.

Operations/data adapters: Tools may interact with the database in two ways. New tools that implement their I/O using the DML can interact directly with the IP. Existing tools and tools not using the DML, however, access data in the database through data adapters. Tools may also need to use information in a form different than that present in the database. This is done by providing mapping functions - data adapters - that transform the information on input and/or output between the desired internal and external forms. These tools can be encapsulated in an *envelope* that provides a data access interface to such tools and invokes these operations transparently. These operations make calls in the DML, access the data in the database and make it available to the tools in the desired form.

There are three kinds of *users* of such a framework: the tool user, the tool builder/integrator and the IP administrator. The tool user uses tools to develop distributed applications. A tool user need only know how to invoke these tools and how to provide them with the appropriate user specified inputs and possibly outputs from the database. Tool builders should know the data definition language and information models to be able to modify the schema and extend the information model - if necessary. They should also use the data manipulation language to write operations that allow the tools to interact directly with the database. An associated *user's manual* tells tool designers:

- What they have to do to develop a tool that can be integrated into this DISE framework.
- How to use the data definition and manipulation languages.
- A description of the current database schema.
- How to do the integration - extend the information model and schema, if necessary.

The IP administrator is responsible for instantiating a schema on a per-project basis and tailoring it specifically for that project. This may involve schema modification (addition/deletion of types, properties, relations etc.), setting access control rights and permissions, etc.

The integrating framework has no specific performance/real-time processing requirements. The absolute performance characteristics of the framework cannot be specified independently of the underlying

hardware/software environment, the processing/time-sharing load on the system, and the user-selected activity being performed; therefore, no direct throughput and response time measurements can be given. However, there is some overhead associated with accessing data from the database over the cost of a direct file access from the file system. This overhead, measured in terms of increased response time or reduced throughput, may be significant.

3.1.4.2. Allocation System Function

3.1.4.2.1. Rationale

Distributed application developers are faced with the question of how to assign objects (the software units of distribution that make up the distributed program) to the processing nodes. While the manner in which they are assigned does not affect the application's functionality, it has major implications for objectives related to performance, fault tolerance, resource utilization and se ty. Assignments vary with respect to how the program performs during execution since resource utilization profiles, communication loads, and parallelism are all influenced by the pattern of allocation. The allocation problem is an important concern in DISE, where multiple processing resources are available and applications consist of multiple objects, each of which can generally be assigned to execute on any processing platform.

Early in development, at the high-level design stage, the developer is concerned with how to decompose the system into individual objects. Later, in developing specifications for objects, operation definitions and object interaction patterns are of importance. At each of these stages, and after coding is complete, the nature of the distributed processing environment has important implications with respect to development decisions. For example, a coarse decomposition derived early in development may not be capable of properly utilizing abundant processing resources. A fine-grained decomposition, on the other hand, may be inappropriate for an environment consisting of just a few processing sites. Hence decomposition is a design time issue with run-time implications that depend on the assignment of units to processing nodes.

Now consider any given decomposition, however obtained, resulting in a particular set of objects. Different assignments of those objects to processing nodes result in different run-time performance profiles for the application, since each assignment necessarily exhibit unique patterns of resource utilization and hence may perform better or worse than other assignments. The allocation problem is an important issue in the development of distributed applications from design through implementation.

The allocation problem, while relevant at a variety of stages, is inherently complex. Given a processing environment consisting of p processors and an application composed of t objects, there are p^t possible assignments of the objects to the processing sites. Furthermore, the problem is known to be NP-Complete, meaning that no polynomial time algorithms for determining an optimal assignment are likely to be found. Despite this complexity, the allocation issue must be addressed in the development of distributed applications in DISE.

3.1.4.2.2. Functionality

The purpose of the Allocation tool function is to provide automated assistance in determining an efficient assignment of units of distribution (objects and clients in the distributed application) to hardware platforms that meets application performance objectives. Distributed application developers may require this Allocation function at a variety of developmental stages. Early in development, high-level program decomposition is impacted by the performance implications of that decomposition, given a particular distributed processing environment and an assignment. At a later stage, when coding is complete, the developer is again faced with the problem of how to assign object modules to processors.

The Allocation tool can be re-employed at this stage of development, since more information about the distributed application is available at that time (in the form of actual code) and on the basis of this additional information, improvements in the allocation scheme may be possible beyond the results of earlier invocations.

The Allocation tool function requires three main parameters:

- A model of the processing environment. This reflects processor attributes such as speed, as well as the communication network topology and associated costs.
- A definition of the distributed application.
- Information about the application's execution behavior.

The Allocation tool generates:

- An assignment of objects to processing nodes;
- The estimated run-time performance under that assignment.

Such information can be utilized, for example, by a program invocation utility that distributes and initiates the appropriate processes (objects and clients) on the designated nodes.

3.1.4.3. Reliability Analysis System Function

3.1.4.3.1. Rationale

One of the most important characteristics of an effective C^3 system is survivability. Survivability is the ability to meet mission requirements in the event of hardware failures and can be measured in terms of reliability and availability. Successful development of survivable applications requires the ability to evaluate their reliability and availability characteristics. This is of critical importance in DISE, which supports the development of C^3 applications for execution in distributed processing environments.

The reliability of an application can be enhanced by various software mechanisms such as atomic transactions, concurrency controls, and replication. Replication in particular can provide the basis for continuous processing in the context of hardware failures. Degradation or termination of the application in the event of failures can be avoided if data and code are replicated on several sites in a system. Replication, however, introduces a significant cost overhead due to maintaining data consistency through distributed concurrency control mechanisms.

Therefore the developer must balance the benefit of application fault tolerance against the inherent cost associated with fault tolerance mechanisms. Furthermore, development decisions concerning reliability made at a later stage of development that lead to unacceptable levels of reliability or performance can result in very costly redesigns. It is imperative that developers have the capability for design-time analysis of reliability characteristics. This supports the construction of distributed applications that meet reliability objectives in a cost-effective manner.

3.1.4.3.2. Functionality

The Reliability Analysis function provides the ability to evaluate reliability characteristics of distributed application designs. This allows developers to build applications that:

- Are partitioned into an appropriate set of communicating objects,
- Meet reliability objectives,
- Incorporate fault tolerance mechanisms only to the extent required to meet stated reliability objectives, thereby controlling cost overhead.

To provide the necessary degree of flexibility, the user must be able to define particular application components or subsystems that are of interest and subsequently measure the reliability characteristics of those components. Reliability characteristics of a distributed application executing in a potentially faulty processing environment are a function of several parameters:

- The reliability of the underlying hardware components,
- The assignment of the software components (including replicated components) to the processing platforms,
- The pattern of interaction (dependencies) between the software components.

These parameters are therefore required input for the reliability analysis function. This System function generates reliability measures for application components. These measures may include but are not limited to the probability of failure by time t and mean-time-to-failure (MTTF) for the specified application components.

3.1.5. System Functional Relationships

The top-level functional relationships between System functions described in subsection 3.1.4 are depicted in Figure 2. The Allocation and Reliability Analysis System functions are functionally related to the Tool Integration Framework function in that both employ the Tool Integration Framework System function for their external data requirements. The arcs in the Figure 2 represent physical data links between the System functions.

The Allocation and Reliability Analysis System functions have a logical relationship in the sense that both support the development of distributed applications and may be employed in a complementary fashion during the development process. However, the functional interface between them is defined in terms of the particular requirements of the application developer and is implemented via the respective interfaces with the Tool Integration Framework function.

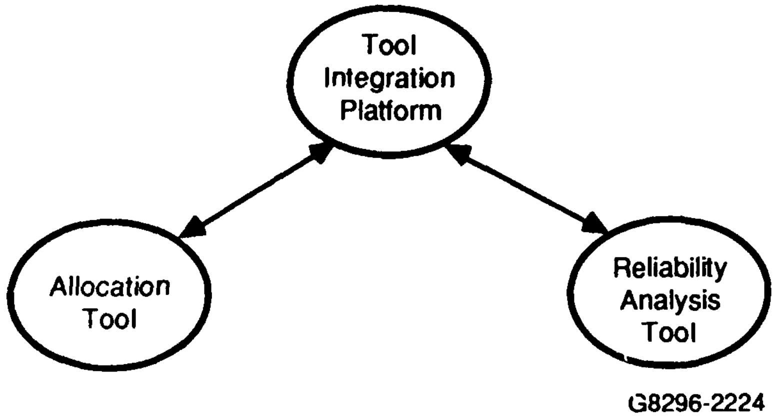

Figure 2. System Functional Relationships

3.1.6. Configuration Allocation

This paragraph presents the detailed functional requirements of the System functions. Each of the System functions described in subsection 3.1.4 corresponds to a single Computer Software Configuration Item (CSCI). The three CSCIs defined below are: Tool Integration Framework, Allocation tool, and Reliability Analysis tool.

3.1.6.1. Tool Integration Framework CSCI

3.1.6.1.1. Functional and Performance Requirements

This section focuses on the second component of the framework, the project database. The high-level requirements of the tool integrating framework - and the project database in particular - are specified below.

- **Data capture:** Data generated and used by the supported software development tools reside in the project database. The tools in the integrated tool set interact with the database for all their I/O needs. Hence data used and generated by tools during software development is captured in the database.
- **Data retention:** The database acts as a repository of data/information where all data associated with the supported tool set is retained.
- **Data provision:** The database provides data to the software development tools upon request.
- **Data consistency:** Integrity and consistency of the data in the database should be maintained. This may involve actions such as processing dependencies to propagate updates, checking access control rights and permissions, etc.
- **Tool integration:** By capturing and storing all the data associated with the tools in a canonical form, the framework provides an integrating mechanism for the tools.

These requirements can be decomposed into the following lower-level requirements.

Information model support: The types of data captured in the Integration Platform database (IP) is determined by studying the input/output requirements of tools: Cronus tools, our candidate tools, and a representative set of existing tools. This study defines types of data used in software development in the DISE environment.

Different tools need different types of data and in different forms. The information model, while not yet complete, must be able to accommodate different types of tools: tools applied at different stages of development and functionally different tools. This effort emphasizes model extensibility - being able to add/model new types of data to accommodate new tools in the supported tool set - rather than model completeness or model sufficiency in accommodating all known existing tools.

Cronus tools such as gendoc, definetype and genmgr interact with, and are to some extent integrated by, the Type Definition Database (TDDB). While existing tools such as these may not be able to use all the features and information in the project database, they are not excluded from using or contributing to that information. These tools must be integrated into the framework without modifying their internals or changing their I/O requirements. Integration of these tools involves being able to capture and retain the data required by these tools in the database, maintaining the necessary relations and providing access to the data. Hence information currently in the TDDB must either be in the database, or support must be provided to allow these tools to interact with the TDDB transparently while executing in this integrated environment, possibly by encapsulating the TDDB as a database object.

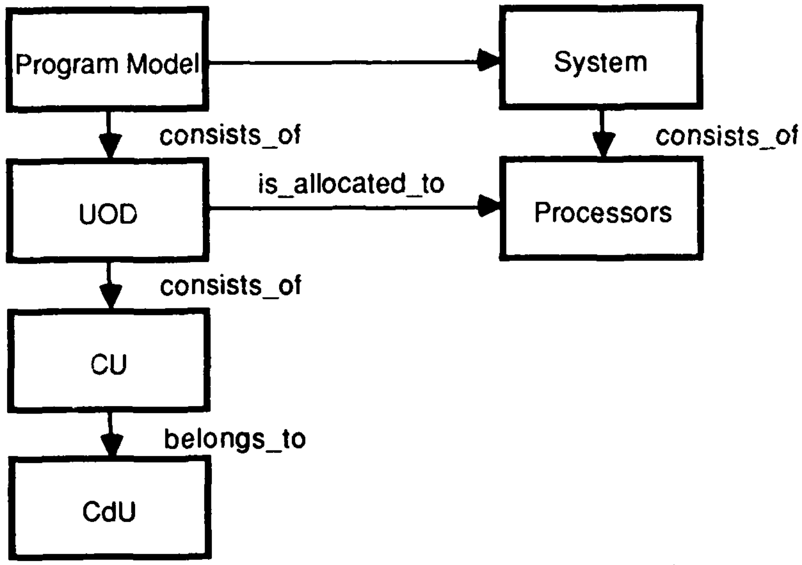

Figure 3. Database Schema Example

For example, the different data types of information required in a distributed program may be:

- A compilation unit (CU) that represents the source code for a program module;
- A compiled unit (CdU) representing compiled object code for that CU;
- Units of distribution (UOD) that are program components allocated to processors in the system;
- Processor data (Processors) giving details about processor speed, name, capacity and memory;
- A program model of the system (Program Model) that uniquely specifies a particular configuration of the system.

Database Schema support: The logical organization of data in the IP is described by a conceptual schema. Data used by the different interacting tools is stored as software entities/objects in the database. The engineering and scientific applications this IP have to support, unlike the traditional database applications, have requirements for numerous data types - types of data items - and relatively fewer instances of these types. Moreover, sufficiently rich modeling primitives are required to describe the intricate structures and interrelationships between the software objects.

Instantiated on a per-project basis, the schema permits the addition of new software types, properties for the types, operations supported by the types, etc. so that it can be tailored for that project's needs. In addition, this extensibility of the IP is a desirable property as new/foreign tools added to the supported tool set may require new software types or new operations.

Hence, the data in the information model is organized into a schema. Some of the requirements of the

schema are that it should:

- Permit the creation/definition of new/different types of software entities.
- Allow the addition of new types (schema extension).
- Provide a data definition language (DDL) to allow the user to create a new object type and specify its placement in the type hierarchy and the operations it implements and inherits.
- Capture the dependencies and relationships among the types in the database by supporting and maintaining relations among them.
- Impose a hierarchical type structure to facilitate inheritance. This makes model extension easier, as the common methods/properties/relations can be inherited by the new type from its parent type.

The example database schema in Figure 3 shows the way the different types in a database may be structured. Figure 4 shows the hierarchy in which these types may be arranged.

Tool support/access capabilities: Tools interact with the data in the IP via operations invoked on the database. Existing tools that currently interface to the native operating system's file system for their I/O can be fitted with indirect database access capabilities. Envelopes can be built around the tools to perform some pre-invocation processing to extract data from the database prior to the tool's execution and some post-invocation processing to update the IP with information resulting from the tool's execution. Figure 5 provides a schematic of the tool envelope used to integrate an existing tool. This

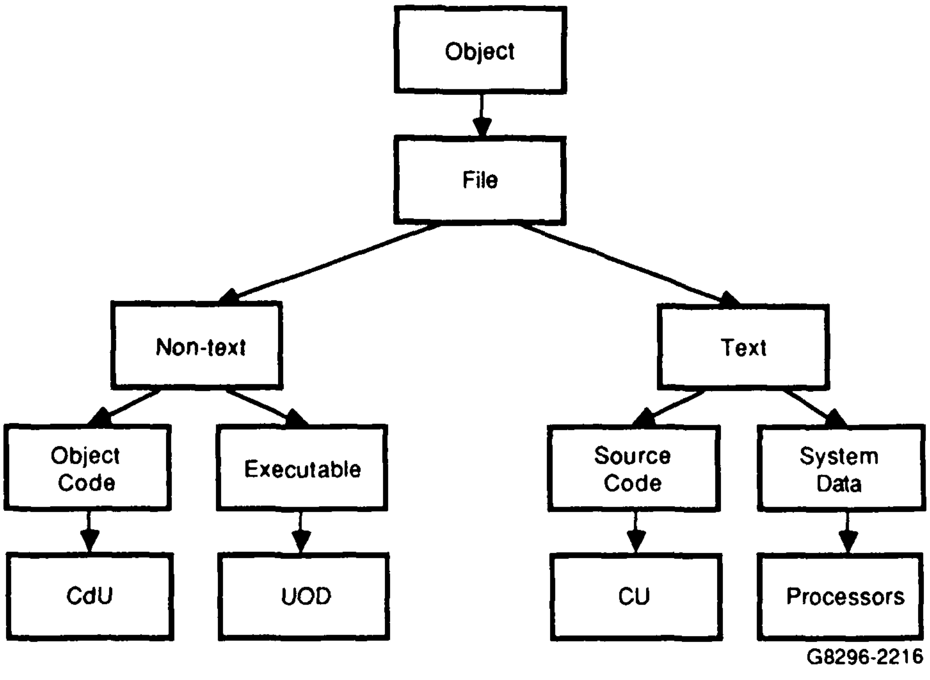

Figure 4. Database Type Hierarchy Example

envelope:

- Extracts the required data items from the database;
- Transforms the data form, if necessary, to be compatible with the tool's input needs;
- Invokes the tool;
- Transforms the tool's output data, if necessary, to be compatible with the IP;
- Updates entities in the IP with the results of the tool's execution.

A direct access capability allows new, independently developed tools to manipulate (access and update) data in the IP directly by interacting with the database via calls made in a data manipulation language (DML). No pre- or post-invocation processing is necessary if the tools are built to interface with the IP for their I/O.

A DML Interpreter in the IP receives these requests, interprets them and invokes the requested operation on the appropriate object. With this DML the user/developer can:

- Create or delete objects (instances of object types present in the schema).
- Access (read and write) data by traversing the schema, locating the desired object, and accessing

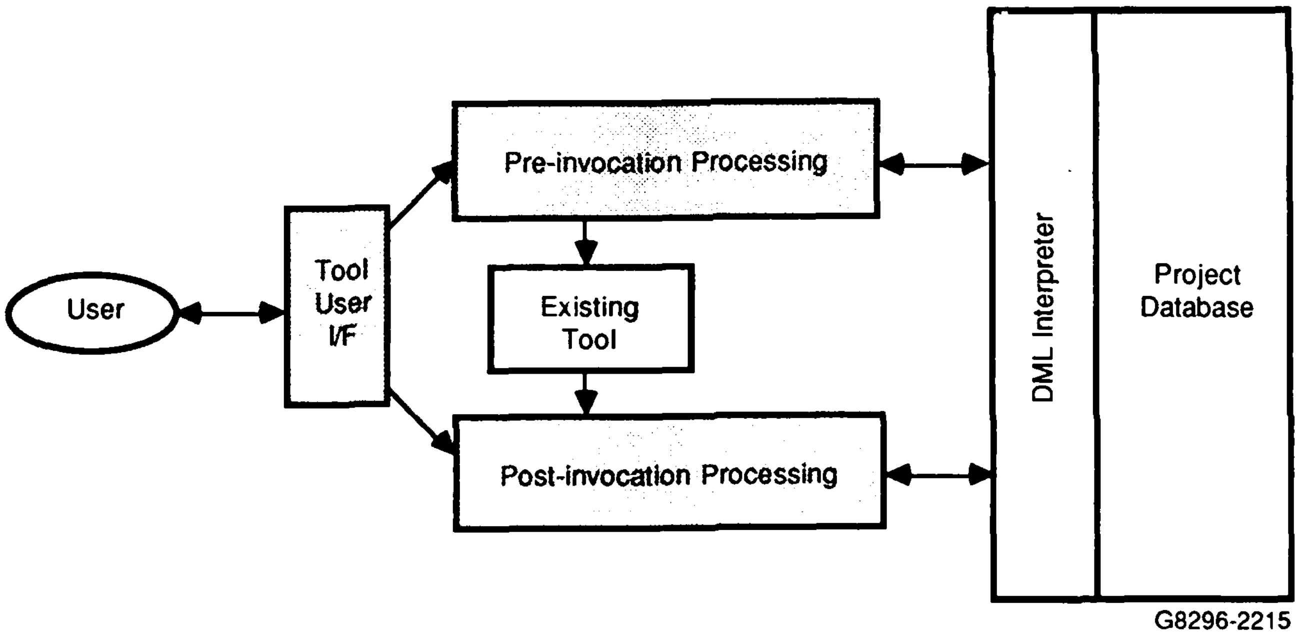

Figure 5. Indirectly Coupled Tools

the data contained.

- Create, delete or modify relations (explicitly) between objects.
- Specify or determine the attribute values (properties) associated with an object.

Type definition and schema extension capability: To tailor a schema to a project's needs, or to accommodate new (and foreign) tools in the supported tool set that may require different types of data, new types may have to be added and the schema modified. This schema extension/modification is done using the DDL. This language facilitates schema extension by allowing one to specify:

- Definition of new types;
- Specification of methods/interfaces to these object types;
- Addition of new operations/properties to both new and existing types.

A type entity hierarchy must be defined for the types in the database. Organizing the types in the database into a hierarchical structure greatly simplifies the task of adding new types to the database by permitting the new types to inherit operations, properties and relations from their parent types.

Configuration management and version control capabilities: This is an important issue in large software system development. It is not explicitly addressed here as it is not directly relevant to tool *integration*. Configuration management and version control tools, however, can be integrated into the framework, as the database schemas can be extended to accommodate such tools.

User Interface capabilities: The main objectives of a user interface management system in the tool integration framework would be to provide:

- Help capabilities to instruct a user/developer;
- Advice capabilities;
- Training instructions;
- Environment/schema instantiation control capabilities that allow the project administrator to specify and instantiate a schema for this project, tailor it to its needs by modifying the schema using the data definition language, provide some initialization information like the list of users in this project, maybe access rights and permissions required for different kinds of access, etc.
- Guidance on applicability of tools: which tool can be applied on this particular object and which object does this particular tool work on.
- Perusal capabilities that would allow a user to look at and traverse the schema instantiated for this project.
- An ad-hoc query facility to ask questions like "Who is the owner of this object ?", "Which objects were created today ?", etc.

While the above functionality is desirable, the time and resource constraints for the development for this CSCI do not permit the construction of such an interface. The user interface to this Tool Integration Framework and the project database permit:

- Ad-hoc queries the user can use to get some basic information about the schema, the types it contains and the values of certain properties of these types;
- Schema instantiation on a per-project basis and the addition/deletion of types to tailor the schema for its specific needs;
- Basic help/advice instructions.

3.1.6.1.2. Requirements Cross-Reference

This CSCI implements all the requirements specified in subsection 3.1.4.1.

3.1.6.2. Allocation Tool CSCI

3.1.6.2.1. Functional and Performance Requirements

The Allocation tool CSCI performs the Allocation System function specified in subsection 3.1.4.2. It determines an efficient assignment of application modules to processing nodes in the system. It can be employed across developmental stages, from early in the design phase through implementation.

The inherent complexity of this problem places some stringent requirements on the structure of the tool. The task of finding a near-optimal assignment is necessarily difficult and potentially time-consuming. Therefore, evaluation of a particular candidate solution must be made quickly since it is anticipated that many alternatives must be examined in order to find a good one. In particular, compiling and executing the application under the candidate assignment to obtain empirical execution data is cost prohibitive. Rather, a cost modeling technique must be employed that generates a reasonably accurate execution cost estimate in a short amount of time.

A schematic diagram for the Allocation tool is given in Figure 6. The tool is composed of three main components: Cost Model Generator, Optimizer, and User Interface. There are four primary inputs to the tool (provided through the User Interface) including software and hardware descriptions and the single output of an assignment and its estimated performance. In the following specification, we describe each input category in the context of each component's description.

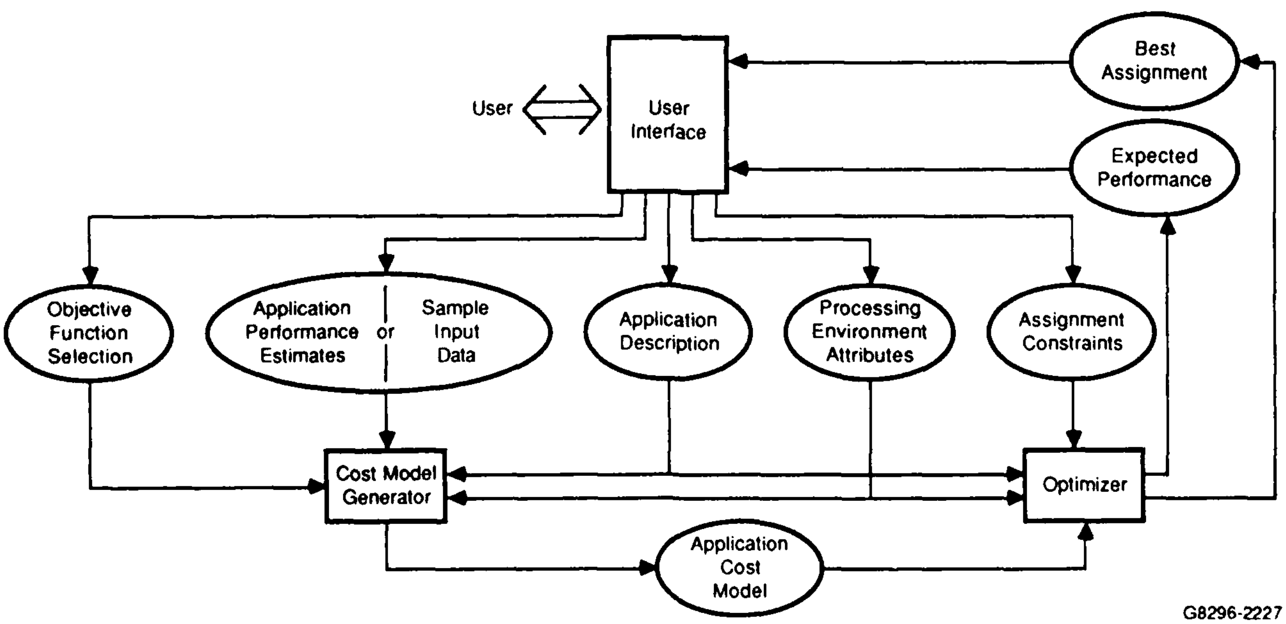

Figure 6. Allocation Tool Schematic

Cost Model Generator: The cost model generator's (CMG) derives an expression representing the distributed application's cost as a function of the assignment of objects to processing nodes. This cost function can then be evaluated in the context of any given assignment, yielding the expected run-time performance under that assignment. Figure 7 shows a detailed schematic of the CMG.

The CMG requires four inputs:

- The distributed application under study,
- Estimated costs of the application components,
- A description of the processing environment,
- An objective function.

The distributed application under study must be provided to the tool. The tool accepts any of three forms, each appropriate at a particular stage of development:

- A set of object definitions without specifications, corresponding to a high-level decomposition;
- A set of object specifications, containing operation headings but without their bodies, corresponding to an intermediate stage of design;
- Fully coded applications in which objects have both specifications and bodies.

The next piece of input data required is the estimated costs of the software components. The application performance modeling is highly dependent upon this information. However, the nature of these estimates vary depending upon the stage of development, becoming increasingly accurate as development proceeds. The three different formats for this information correspond to the three different application input forms above:

- Object workloads,

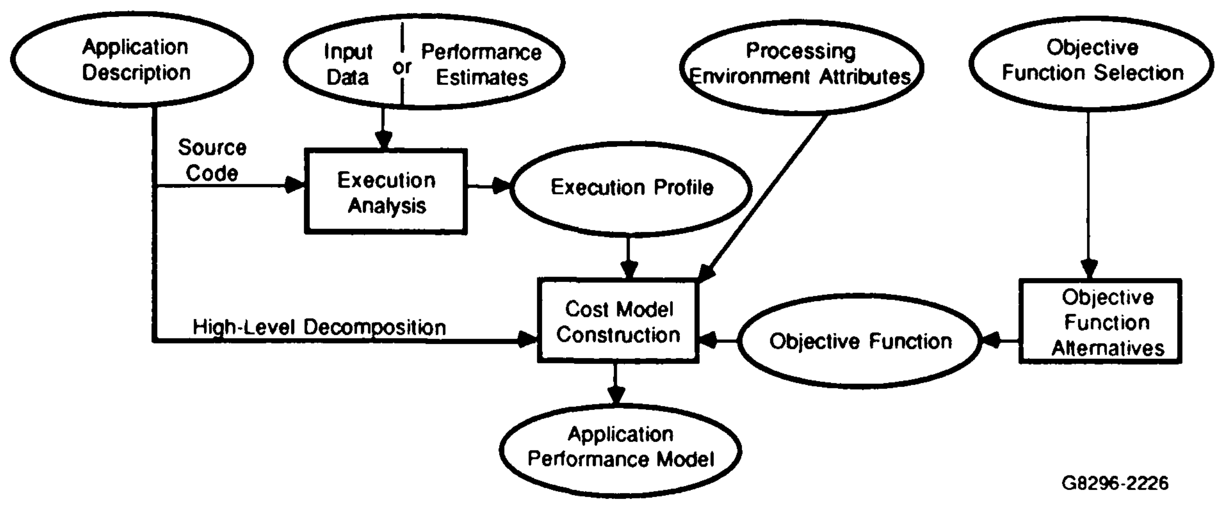

Figure 7. Cost Model Generator Schematic

- Operation costs and object interaction profiles,
- Application input data.

When the first form is used early in design, these costs are estimated workloads for each object. The workloads correspond to the total computational load each object is expected to place on the system during the execution of the application.

With the second input form, when specifications are complete, cost estimates are required for each operation within each object. These correspond to the expected cost of an arbitrary invocation of the operation within the object, not its total workload. In addition, estimates must be provided of how many calls each object makes on operations within other objects. These communication patterns may be captured in a simple matrix.

The cost estimates associated with the first two input forms must be provided directly by the user. No other source of run-time behavior information is available to the tool at these earlier stages of development. For the final input form, when coding is complete and source code is provided as input, user-supplied application cost estimates are not required. Instead the user simply provides representative sample input data as defined by the particular application.

The next input item required by the CMG describes the processing environment and includes:

- Processor attributes,
- Communication network parameters.

The relative speed of each processor is required input. In addition, special attributes of processing sites must be identified (for example, memory capacity, special devices, and additional processing resources such as a floating-point co-processor). This information affects the notion of a "best" host for each application module.

The processing environment description must also include cost models for the communication system. In particular, a cost model reflecting the cost of data transmission is required, as well as a cost model that defines the nature of communication system delays. This latter model may be a complex function of the allocation of objects to nodes, the total message traffic, and a profile of communication events as they occur over time. This last quantity is extremely difficult to determine since it depends on the actual allocation scheme for the application, and therefore only the first two quantities are utilized. While this is less than ideal, it is necessitated by the complexity issues identified in subsection 3.1.4.2.

Finally, the CMG requires an objective function. Two objective functions are available and selectable by the user:

- Application response time or throughput,
- Total execution time for the application.

Response time is the wall-clock measure of program execution time, while total execution time is the summation of execution costs across host processors. The distinction between the two is that total execution time does not take into account the possibility of parallel execution of different components. Which measure is more relevant depends on the particular requirements of the application developer.

The inputs just described can all be provided by the user directly. However, the only necessary inputs from the user are the application and its associated cost components. In general, the processing environment parameters are previously defined for the target environment.

Given these input items, the CMG creates a cost function describing the expected performance of the application as a function of a given assignment. The primary terms in the model, under both objective

functions, are:

- Execution cost for each object,
- Communication frequency between object pairs,
- Cost for communication events,
- Communication delay costs.

Additional terms reflect the effect an assignment has on certain of the above terms. For example, two communicating objects allocated to the same processing platform reduces the magnitude of certain communication costs, as the communication network is not used under those circumstances.

Generally, construction of these terms on the basis of the specified input data is straightforward. However, use of the third application input form (source code along with sample input data) necessitates additional computational effort in the CMG in order to derive execution and communication cost terms. Syntactic analysis of source code is of limited effectiveness in this context. Therefore, this analysis is performed by compiling an instrumented version of the distributed application and executing it (perhaps on a single site) to obtain execution profiles for each operation and to determine the frequency of communication between each pair of objects. This dynamic program analysis function is a major component within the CMG. Construction of the communication event cost term is based on the cost of communication as implemented in the host language. Communication delay costs are necessarily based on the underlying communication network, and therefore must be reflected in the processing environment attributes.

Optimizer: The Optimizer's function (see Figure 8) is to find a good assignment of application modules to processing nodes. Goodness is defined by the application cost model obtained from the CMG. Independent of the accuracy of that cost model, the optimization task is extremely difficult.

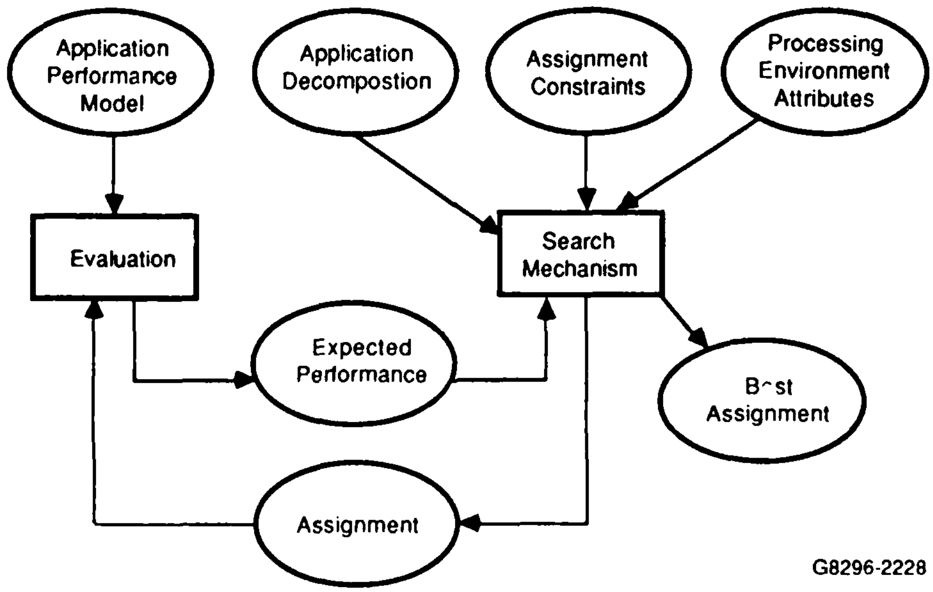

Figure 8. Optimizer Schematic

The task can be defined as a search in a combinatorially large solution space, where each dimension corresponds to the possible processor assignments for a particular object. The performance associated with a particular point (a mapping of objects to processors) is defined by the application cost model. The Optimizer includes an evaluation component, seen in Figure 8, which simply evaluates the cost model in the context of a given assignment (a point in the solution space).

The inputs to the Optimizer include:

- The application decomposition,
- The hardware environment description,
- The application cost model (from the CMG),
- Search constraints.

The application decomposition, also required by the the CMG, is needed simply to identify the software components to be allocated. The hardware environment description is used similarly, to identify the host processing platforms in the system. The search constraints are optional user-specified constraints that bound the search space for optimization. For example, the user may specify that one particular object must be assigned to a certain subset of nodes; or a certain object might have to be replicated across a specified number of nodes, or a certain processor allowed to host a limited number of objects. Accepting this type of input facilitates experimentation by the user.

Given these inputs, the Optimizer attempts to find a *good* point in the large, complex search space defined above. There are three primary requirements for the Optimizer:

- Effectiveness,
- Robustness,
- Efficiency.

Clearly, the Optimizer should be effective in that it should be capable of finding an assignment of objects to processors that has reasonably good expected performance. This eliminates consideration of random-walk search techniques, for example, since they generally cannot be expected to find a reasonable solution in a reasonable amount of time.

The optimization technique must also be robust. The solution space is expected to contain numerous local optima. These represent optimal solutions over a narrow region of the solution space but which may in fact be vastly inferior to solutions in neighboring regions. Robustness implies that the optimization technique must not fixate on these points. Hill-climbing techniques, so named for their ability to find these local optima, would require modification to meet this criteria.

Finally, the technique must be reasonably efficient. Due to the inherent complexity of the problem, however, consumption of significant computational resources may be necessary. This implies that interactive performance may not be achievable. On the other hand, it does eliminate exhaustive enumeration algorithms and in general any algorithm exhibiting exponential (or high-degree polynomial), worst-case performance. To provide a high degree of flexibility with respect to the potentially high cost of optimization, computational limits on the Optimizer can be specified by the user in either of two forms:

- Maximum time for search,
- Termination criteria.

The first form places bounds on how long the Optimizer can search. The second specifies acceptable performance goals - the expected performance of an assignment may be within user-defined tolerance limits while being suboptimal.

Many algorithms developed for this type of search have been described in the literature and meet the above criteria to varying degrees. In this CSCI, several algorithms may be used to provide greater generality and flexibility. A greedy algorithm that quickly finds a reasonable, although non-optimal, solution may be appropriate when the input is just the high-level design (the first application input form). The cost estimates at this stage are rough; therefore it is not cost-effective to spend a large amount of time employing a powerful search technique. On the other hand, when source code is provided as input, the module cost estimates are more accurate, and therefore use of a more powerful, computationally intensive search technique becomes appropriate. The choice of technique is also influenced by the user, as noted above, in that the user may specify bounds on the search time that eliminate the use of certain algorithms.

User Interface: The User Interface component provides template-based facilities that permit the user to enter the various tool parameters. The parameters, discussed in detail above, specified at the User Interface are:

- Application identification: The user identifies the application of interest and the User Interface obtains relevant information (source code) from the project database.
- Processing environment attributes: At the user's discretion, these attributes are either provided by the user directly or they may be obtained indirectly through a reference into the project database to a known, previously defined processing environment.
- Application performance data: If complete application source code does not exist for the identified application, performance estimates for application components are provided. This can occur either interactively through templates or through the project database if such data has previously been specified for the application components.
- Application input data: If source code exists for the application, representative input data for it must be provided. Again, this item can either be supplied directly by the user (by specifying input files) or through a reference to an input data entity in the project database.
- Objective function selection: The user must choose an objective function from a list of alternatives provided by the User Interface.
- Assignment constraints: The user may, if desired, specify any number of constraints on the assignment, either in terms of objects or processors. Constraints may also include maximum allowable search time and termination criteria expressed as a level of acceptable expected performance.

Finally, the User Interface provides facilities for displaying the results of a search. At the user's discretion, these results may be saved in a file or in the project database.

3.1.6.2.2. Requirements Cross-Reference

This CSCI implements all the requirements specified in subsection 3.1.4.2.

3.1.6.3. Reliability Analysis Tool CSCI

3.1.6.3.1. Functional and Performance Requirements

The Reliability Analysis tool CSCI performs the Reliability Analysis System Function specified in subsection 3.1.4.3. It provides design-time assistance in measuring reliability characteristics and in evaluating reliability impacts of replication of a distributed application executing in a faulty processing environment.

A variety of factors influence the reliability of a distributed application. The reliability of processing platforms and the communication network play primary roles. But how the hardware reliabilities

influence the application's survivability depends on how the component software modules (objects) are assigned to those processing hosts. If no objects are assigned to a particular host processor, the reliability of that host does not affect the reliability of the application. Furthermore, the pattern of interaction between objects affects reliability measures. If two objects do not communicate, their assignment to two hosts connected by a highly unreliable communication link will not necessarily lead to low application reliability. This tool allows the developer to experiment with different designs by manipulating these parameters, for example, by changing the assignment, changing the level of granularity of objects, modifying hardware reliabilities (optimistically or pessimistically), or changing the interaction pattern.

The Reliability Analysis tool user inputs are described below, and are followed by descriptions of each tool component. A schematic diagram of the tool appears in Figure 9.

Inputs: Five input items are required by the tool:

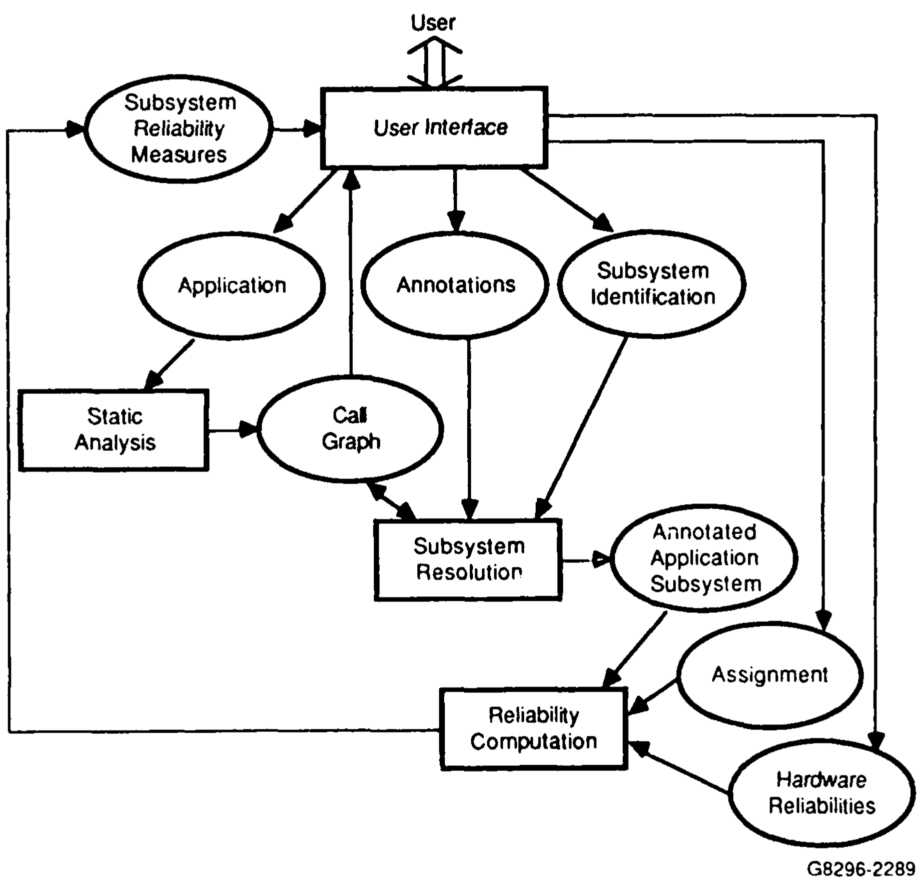

Figure 9. Reliability Analysis Tool Schematic

- The distributed application,
- Hardware reliability characteristics,
- An assignment of the application components (objects) to the processing nodes,
- The selection of a subsystem for investigation,
- Annotations for that subsystem.

The distributed application input item specifies the objects comprising the application, but does not necessarily include any code for the bodies of those objects. However, the description must define dependencies indicating which objects interact directly, that is, which objects are called by each object.

The second input item, hardware reliabilities, specifies the reliability attributes of each hardware element, such as processing units, devices, and communication links. This can be expressed as mean-time-to-failure, for example.

The third input item indicates the host processing platform for each object in the application. A replicated object has more than one processor assignment.

The final two input items provide a means for the user to focus on a particular object or subset of objects as desired. The annotations apply to the dependencies between objects and indicate the probability of interaction between the objects. Given an object dependent upon another that is located on a highly unreliable processor, for example, a low probability of interaction would dampen the otherwise negative effect brought by the faulty processor on the reliability of the dependent object.

Static Analysis: The Static Analysis component of the tool (see Figure 9) builds a call graph model of the application on the basis of the application description and dependency information. The generated call graph is fed to the Subsystem Resolution component as well as being provided back to the user through the user interface component.

Subsystem Resolution: This component processes the call graph by pruning it to obtain the subsystem specified by the user. For example, if the user selects a particular object for study, the call graph is traversed to obtain a graph consisting of only those objects directly or indirectly called by the specified object. The annotations then apply to dependency arcs in this subgraph. The resulting annotated subgraph is then provided to the Reliability Computation component.

Reliability Computation: Given the annotated subsystem of interest, the assignment of objects to processing platforms and hardware reliabilities, reliability measures for the application subsystem are computed by this component.

The reliability characteristics of an application can be expressed in terms of certain probabilistic measures such as availability and mean-time-to-failure (MTTF). The availability [Barlow81a] of a component is a function of time indicating the probability that the component is functioning at any given time. It is dependent on the availability of the paths to the required resources, the sites holding the resources, and the sites executing the components. The reliability of a component may also be expressed as a probability indicating that a component has not failed during a specified time interval. The mean-time-to-failure (MTTF) for a component in a distributed system is the expected time interval during which that component remains available before a failure occurs. A component fails if it is unable to access any of the required resources, or if the node executing the component fails. The Reliability Analysis tool generates one or more of these reliability measures at the user's discretion.

A considerable amount of work has been done in the evaluation of reliability and availability of paths in communication networks. Reliability analysis techniques such as NetRAT [Wang83a] address the problem of pair-wise terminal reliability in communication networks. The technique concerns the availability of paths from a set of nodes in the network to a different set of nodes. Other work on reliability approximation has been done by reduction to a serial-parallel graph model of the system

[Sahner87a] using directed acyclic graphs. Simulation can also be employed to obtain reliability estimates. The Reliability Analysis tool evaluates the reliability of a design based on one or a combination of these techniques.

User Interface. This component provides facilities for entering relevant parameters. Some are simply identified by the user and obtained from the project database.

- Application identification: The desired application is named by the user. Component objects in that application are retrieved from the project database along with object dependencies.
- Hardware reliability attributes: After the user names the target processing environment, reliability attributes are retrieved from the project database (if they have previously been specified).
- Assignment of application to hardware: This information can be obtained directly from the project database, as it represents a relationship between two database entities. The Allocation tool, for example, generates this information. If the information is not available, the user must supply it.
- Call graphs: After being generated by the Static Analysis component, the call graphs are made available to the user through the User Interface for browsing purposes. This facilitates subsystem selection and annotation.
- Subsystem selection and annotation: The User Interface, using the call graph model, provides convenient means for the user to select an object or subsystem of interest. The resulting subgraph is displayed, and the user is provided a suitable means for annotating arcs in the graph.
- Subsystem reliability measures: The generated reliability measures for the specified subsystem are displayed to the user. At the user's option these measures can be incorporated back into the project database as attributes of distribution units.

3.1.6.3.2. Requirements Cross-Reference

This CSCI implements all the requirements specified in subsection 3.1.4.3.

3.1.7. Interface Requirements

3.1.7.1. Interface Identification

A variety of interfaces exist in the context of this System. There are interfaces between the three CSCIs defined in paragraph 3.1.6 and the host distributed operating system, Cronus. The Tool Integration Framework must provide interfaces with programming tools in DISE. These tools form two classes: new tools designed and built to use the Tool Integration Framework directly, of which the Allocation and Reliability Analysis tools are examples, and existing/foreign tools which were not built in the context of the Tool Integration Framework.

3.1.7.2. Cronus to CSCI Interfaces

3.1.7.2.1. Cronus to Tool Integration Framework CSCI Interface

The external interfaces of this CSCI include its interface to the underlying Cronus distributed operating system. The following are some of the requirements of this interface.

- The IP has to be incorporated in Cronus, without modifying the internals of Cronus.
- The IP has to be accessible to all the tools in the supported tool set.

- Location transparency for the IP should be provided.
- Cronus must provide support for any operation that can be invoked on the data in the IP.
- The integrity and consistency of the data in the database are essential.
- Recoverability and persistence of data are desirable.

Encapsulating the Integration Platform in a primal Cronus object provides one way of meeting some of these requirements. A Cronus system manager can be created to manage and create object instances to correspond to project-specific instantiations of the database schema. Generic operations, to permit the creation of objects, and non-generic operations, to permit access to the data types in the Cronus object, can be provided. The I/O request of a tool results in a request to the IP manager which routes this request to the appropriate IP object. The appropriate operations to access the data in the IP are then be invoked.

There are certain advantages and disadvantages to this approach. The existing Cronus object creation and message routing facilities and the mechanisms available for generic and non-generic operations can be fully utilized. These operations ensure data consistency and integrity, by providing consistent access to the data. Persistence of data is possible to the extent that Cronus objects can be recovered.

However, implementing the IP as a Cronus manager also imposes certain constraints. If the IP manager creates an object for each schema supported, the information model of which these schemas are instantiations also must somehow be incorporated in this representation. While Cronus TDL is be used to write and create the IP manager, DDL is used to tailor the information model on a per-project basis and instantiate schemas. As Cronus objects are passive, the IP can only respond to explicit requests and cannot do any background processing. Moreover, as only one thread of control can exist at a time, no concurrency is possible. If a project-specific database schema is represented as one Cronus object, routing of requests to particular data objects *within* this schema have to be done explicitly, as Cronus routing facilities are inadequate for this.

3.1.7.2.2. Cronus to Tool CSCI Interfaces

Requirements for the interface between the two CSCI tools and Cronus is not specified.

3.1.7.3. Tool Integration Framework CSCI to Tools Interfaces

All tools in the supported tool set interact with each other via the Integration Platform of the Tool Integration Framework. There are two kinds of interfaces to the IP for the two kinds of tools: indirect coupling for existing/foreign tools and direct coupling for new tools.

The DML Interpreter acts as the data access manager, interpreting the DML calls and invoking the requested operations on the data objects. These DML calls allow the tools to:

- Traverse the schema.
- Access data objects - read/write data.
- Update data
- Reserve/lock data objects
- *Import* and *export* data between the database and the tools.
- Create and delete instances of these objects.
- Read/specify properties in the objects.

The issue of data local to a subset of tools in the supported tool set must be considered. For example, existing Cronus tools like gendoc, genmgr, etc. store all their relevant data in the type definition database (TDDB). This database hence serves to integrate these tools. These tools have to be integrated

into the framework without modifying the internals of these tools or changing their I/O requirements. Integration of these tools involves being able to capture and retain the data needed by these tools in the database, maintaining the necessary relations and providing access to the data. Hence, information currently in the TDDB must either be in the database, or support provided to allow these tools to interact with the TDDB transparently while executing in this integrated environment by possibly encapsulating the TDDB as a database object.

Subsections 3.1.7.3.1 and 3.1.7.3.2 describe these interfaces between the Integration Platform (IP) and new and existing/foreign tools.

3.1.7.3.1. Tool Integration Framework CSCI to New Tools Interface

3.1.7.3.1.1. Tool Integration Framework CSCI to Generic New Tool Interface

New tools interface directly with the IP by invoking calls made in the data manipulation language (DML). No pre- and post-invocation processing to transform data representations is necessary if these tools are developed so that their I/O requirements can be met by the existing information model and they interact with the IP for their I/O needs.

DML calls can be embedded in the code of these new software development tools to access the IP for data. These calls are interpreted by the DML Interpreter in the IP and processed.

3.1.7.3.1.2. Tool Integration Framework CSCI to Allocation Tool CSCI Interface

The User Interface component of the Allocation tool interfaces with the Tool Integration Framework to obtain data from and add data to the Integration Platform. The bandwidth of the interface is significant and can greatly simplify the use of the Allocation tool. The User Interface component presents to the user, in effect, a highly structured interface into the IP. By simply identifying the application and the processing environment (if they both exist in the IP), the user can employ the full functionality of the Allocation tool.

Input data which can be obtained from the IP includes:

- Application source code entities,
- Performance attributes associated with the application entities,
- Processing environment information including processor entities and their attributes, and a communication network description.

Data generated by the Allocation tool can be incorporated back into the IP for use by other tools. The assignment of application modules to processors can be maintained as a relationship between these two entity types. Its associated expected performance would be recorded as an attribute associated with that relationship.

3.1.7.3.1.3. Tool Integration Framework CSCI to Reliability Analysis Tool CSCI Interface

The Reliability Analysis tool interfaces with the IP at two different levels. The data used and generated by the tool is present in the IP. In addition, the user, via the User Interface component of the tool, is able to retrieve data from the IP, to either peruse it or input it to the tool.

Input data obtained from the IP includes:

- The distributed application; objects representing the units of distribution that constitute this application and relations representing dependencies.
- Hardware reliability characteristics; objects that specify reliability information about the proces-

sors in the system (if they have been previously defined).

- A description of the processing environment; the processor objects representing the processing nodes in the system.
- An assignment of the application's components to the processors in the system. Relations between the units of distribution and the processors to which they are assigned.
- After being generated by the Static Analysis component of the tool, this is made available.

The reliability measures generated by this tool may be represented as attributes of the *program model* or *UOD* objects giving an estimate of the reliability of either the system or a particular program component, given a particular assignment.

Data generated by the tool is placed in the IP for use by other tools or subsequent perusal from the user interface of this tool. For example, the following scenario illustrates how the Reliability Analysis tool and the Allocation tool may interact. Performance impact of the replication mechanism can be analyzed by interfacing the Reliability Analysis tool with the Allocation tool. Replication may improve performance. Locality of reference may result in improved response times. However, performance may suffer due to additional cost in updating multiple copies of the data. Tradeoffs between the reliability and performance with respect to the degree of replication must be considered. The required data is easily accessible and these accesses and updates are kept consistent by the IP.

3.1.7.3.2. Tool Integration Framework CSCI to Existing Tools Interface

Existing tools interact indirectly with the database through operations and data adapters. These tools that currently access data from the native operating system's file system do not use the data manipulation language (DML) to access data. Hence, they are encapsulated by an *envelope*, as described in subsection 3.1.6.1.1, that allows them to access data in the IP.

This *envelope*:

- Extract the required data items from the database. As part of pre-invocation processing, the user defined parameters to this invocation of the tool have to be interpreted, and the desired data items accessed from the database. Operations to traverse the schema (if necessary) and access the required objects are invoked by the envelope.
- Transform the data (if necessary) to be compatible with the tool's input needs. Operations may have to transform/map data between the internal (as present in the database) and external (as required by the tool) forms.
- Invoke the tool.
- Transform the tool's output data (if necessary) to be compatible with its representation in the IP.
- Update (write to the appropriate entities, create relations, specify properties) the IP with the results of the tool's execution. Operations to traverse the schema (if necessary) and update the correct objects are invoked by the envelope.

The envelope, therefore, contains calls to operations which access and manipulate the data. These operations translate the tool's I/O requests to calls in the DML, perform any data transformations/mappings between the desired forms necessary, and make the appropriate calls to the IP.

Figure 5 illustrates the functions of a tool envelope. To demonstrate how an existing tool would be integrated with the IP, let us consider an instrumentation tool. This tool monitors the execution of an application, collecting data (event driven or time driven) and can either display it or store it in a file for off-line analyses. *Hooks* in the application's code make calls to the monitor process to send data, synchronize system events, begin and end measurements, etc. The monitor process collects the data,

organizes and stores it and can perform different kinds of statistical analyses. The results can either be displayed or stored in a file for later perusal or use by some other tool.

This tool can be integrated with the framework by encapsulating the monitor in an envelope so that its existing I/O requests are transformed to transparently access data from the IP instead of the native operating system's file system. This tool interfaces with the IP to:

- Access and manipulate monitor data (maybe for off-line analyses);
- Store the collected data;
- Store the results of analyses performed.

Hence the envelope provides operations to retrieve the data, transform its representations, if necessary, between the desired forms and store the data.

The IP presents the monitor tool with a consistent way of storing and retrieving the generated data and results. The user interface component of this tool and other tools that may need to interact with this tool can, therefore, readily and consistently access the required data from the IP.

3.1.8. Government-Furnished Property List

Government-Furnished Software: Cronus Releases 1.2, 1.3 and 1.4.

A set of Cronus reference manuals is provided with the system. They are:

- CRONUS Installation Manual,
- CRONUS User's Reference Manual,
- CRONUS Tutorial Documents,
- CRONUS Programmer's Reference Manual,
- CRONUS Operator's Reference Manual,
- CRONUS Release Notice.

3.2. System Characteristics

The deployment requirements of this system are that the distributed operating system support be Cronus Release 1.3 running on Sun UNIX 4.0. This coincides with the environment in which the system is developed and in which the system is demonstrated.

3.3. Processing Resources

3.3.1. Host Computer Processing Resource

3.3.1.1. Computer Hardware Requirements

The system and its functions as described in this document are implemented on top of the Cronus distributed operating system, and therefore acceptable computing platforms are those which host Cronus. The experimental implementation proposed for this system uses Sun-3 workstations running Sun UNIX 4.0 hosting Cronus. The CSCIs require Sun hosts only to the extent that the CSCI user interfaces rely on graphics capabilities provided on the Sun Workstations.

3.3.1.2. Programming Requirements

- Programming Language: The C language is used for the implementation of all the CSCIs.
- Compiler: Compilation of the CSCIs requires a C language compiler.

3.3.1.3. Design and Coding Constraints

The CSCIs are developed using a Spiral software process. The Spiral process will be tailored for developing software that is intended to demonstrate the feasibility of tool integration and automate allocation and reliability analysis for distributed applications.

3.3.1.4. Computer Processor Utilization

Not applicable.

3.4. Quality Factors

This effort is a feasibility study of the tool integration framework and the two tools. It culminates in the development of a prototype and a demonstration of its capabilities and the concepts involved.

3.4.1. Reliability

The system will be delivered as a prototype with no known software bugs. Moreover, this system will be implemented on the Cronus distributed operating system, and hence its reliability will be contingent upon the reliability of Cronus.

3.4.2. Modifiability

3.4.2.1. Maintainability

There are no commitments at this point to maintain the system after delivery to RADC. (Refer to subsection 3.5).

3.4.2.2. Flexibility and Expansion

The system can be easily expanded, i.e., new tools can be integrated into the Tool Integration Framework. This is facilitated by a framework that is extensible and database schema that can be modified/extended. New tools are integrated by extending the information model (if necessary) and database schema using the DDL to accommodate (if any) the new types required by these tools. Tools can either be directly coupled or envelopes built around them to allow them to access data in the IP.

3.4.3. Availability

Not applicable.

3.4.4. Portability

Not applicable.

3.4.5. Additional Quality Factors

Not applicable.

3.5. Logistics

3.5.1. Support Concept

There are no requirements that Honeywell support the three CSCIs, as specified in subsection 3.1.6, beyond January, 1990. After that date, further funding would be required from RADC for continuing support. However, an integral component of the deliverables is the *Software Programmer's and User's Manuals* which articulate how to manage the project database and fully documented source code for the CSCIs.

3.5.2. Support Facilities

Not applicable.

3.5.3. Supply

Not applicable.

3.5.4. Personnel

The Tool Integration Framework CSCI requires one or more individuals familiar with basic database concepts. This individual serves as the database administrator and interfaces with tool developers regarding tool data requirements. Routine maintenance and control requires one or more individuals experienced in software development in DISE.

No personnel requirements are anticipated for Reliability Analysis and Allocation tool CSCI maintenance.

3.5.5. Training

Maintenance manuals for the Tool Integration Framework CSCI are included with the delivery of the software. Informal training in the maintenance and use of the Tool Integration Framework CSCI will occur at the time of installation and demonstration at RADC. Informal training in the use of the two tool CSCIs will likewise occur at the time of installation and demonstration at RADC.

3.6. Precedence

The requirements in this document take precedence over descriptions in the First Interim Technical Report.

4. Qualification Requirements

4.1. General

This section specifies the methods to be used to test that the System requirements presented in section 3 have been met to the degree stated in section 3.

4.1.1. Philosophy of Testing

The System is to be tested using conventional testing methods: running the implemented tools and integration framework against test cases which cover usual and extreme inputs to the software.

4.1.2. Location of Testing

Testing is to be performed at the contractor's location.

4.1.3. Responsibility for Tests

The developer is responsible for CSCI level testing and integration testing.

4.1.4. Qualification Methods

The software will be qualified by a demonstration to the funder.

Validation of the Tool Integration Framework is constrained by the availability of the DISE environment and the tools that it integrates. Framework and tools will be validated using the version of DISE installed at Honeywell, SSDC.

Validation of the Allocation and Reliability analysis tools is constrained by the availability of distributed applications under development. Small example distributed applications will be developed and used in validating both tools.

4.1.5. Test Levels

The contractor will perform unit CSCI tests, integration testing, and a demonstration.

4.2. Formal Tests

Not applicable.

4.3. Formal Test Constraints

Not applicable.

4.4. Qualification Cross-Reference

Not applicable.

5. Preparation For Delivery

The System software, including source code and executable code, will be delivered on 1/4-inch magnetic tape in *tar* format. As detailed in the proposal, the documents and manuals will be on 8 1/2 by 11-inch paper.

6. Notes

6.1. References

Penedo88a.

M. H. Penedo and W. E. Riddle, "Guest Editors' Introduction: Software Engineering Environment Architectures," *IEEE Trans. on Software Engineering* **14**(6) pp. 689-696 (June 1988).

Chikof88a.

Elliot J. Chikofsky, "Software Technology People Can Really Use," *IEEE Software* **5**(2) pp. 8-10 (March 1988). (Guest editor's introduction to the special issue on CASE)

Boehm83a.

Barry W. Boehm, "Seven Basic Principles of Software Engineering," *The Journal of Systems and Software* 3(1) pp. 3-24 (March 1983).

Boehm86a.

Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *Software Engineering Notes* **11**(4) pp. 14-24 (August 1986). (Special issue for the International Workshop on Software Process and Software Environments.)

Barlow81a.

R. E. Barlow and F. Proschan, *Statistical Theory of Reliability and Life Testing*. 1981.

Wang83a.

P. Wang and A. Tripathi, *NetRAT: A Network Reliability Analysis Tool*. 1983.

Sahner87a.

Robin Sahner and Kishor Trivedi, ''Performance and Reliability Analysis Using Directed Acyclic Graphs,'' *IEEE Transactions on Software Engineering* **Vol SE-13, No. 10**(October 1987).

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.